

Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models

Zhijie Wang¹, Zijie Zhou², Da Song¹, Yuheng Huang³,
Shengmai Chen⁴, Lei Ma³, Tianyi Zhang⁴

¹University of Alberta, Edmonton, AB, Canada

²University of Illinois Urbana-Champaign, Champaign, IL, USA

³The University of Tokyo, Tokyo, Japan

⁴Purdue University, West Lafayette, IN, USA

*2025 IEEE/ACM 47th International Conference on Software Engineering
(ICSE).*

概要

LLMはコード生成において前例のない能力を示している

LLMによるコード生成のエラーに関してはまだ理解が限られている

そこでこの研究ではHumanEval上の6つの代表的なLLMにおいてLLMのコード生成エラーの詳細な分析を行った

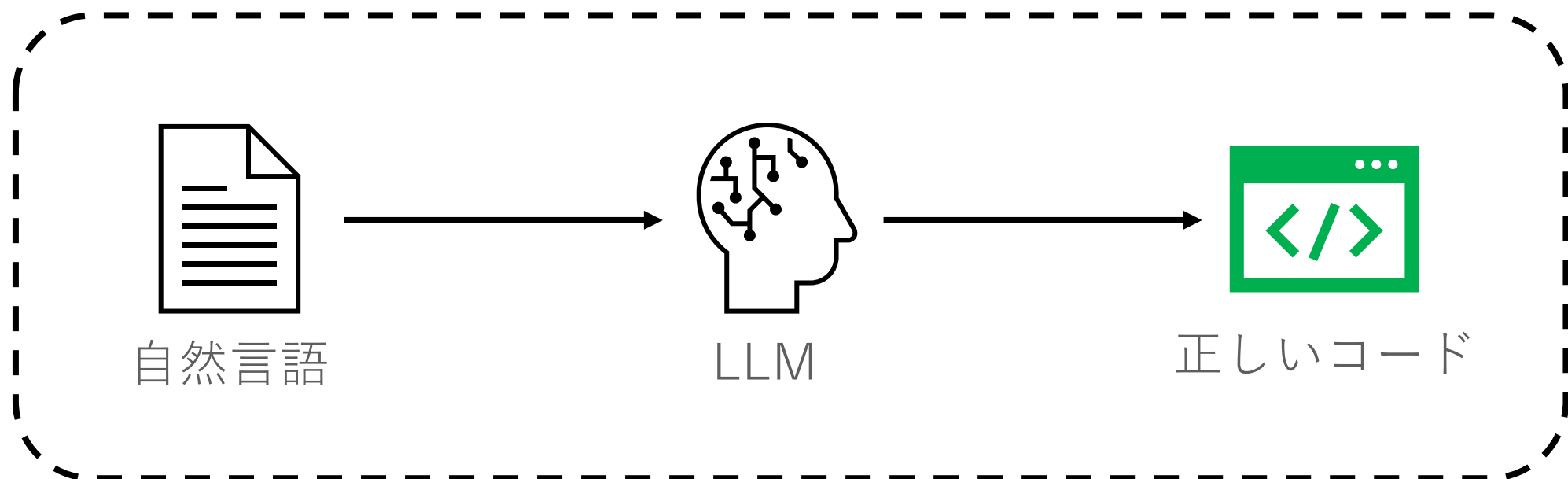
- オープンコーディングとテーマ分析を用いた、コード生成エラーの包括的な分類
- エラーとタスクの複雑さ、およびテストの合格率との相関関係の分析

分析で得られた課題に対処するためのいくつかの将来の方向性について議論した

背景

LLMの進歩により、コード生成タスクは急速な発展を遂げている。

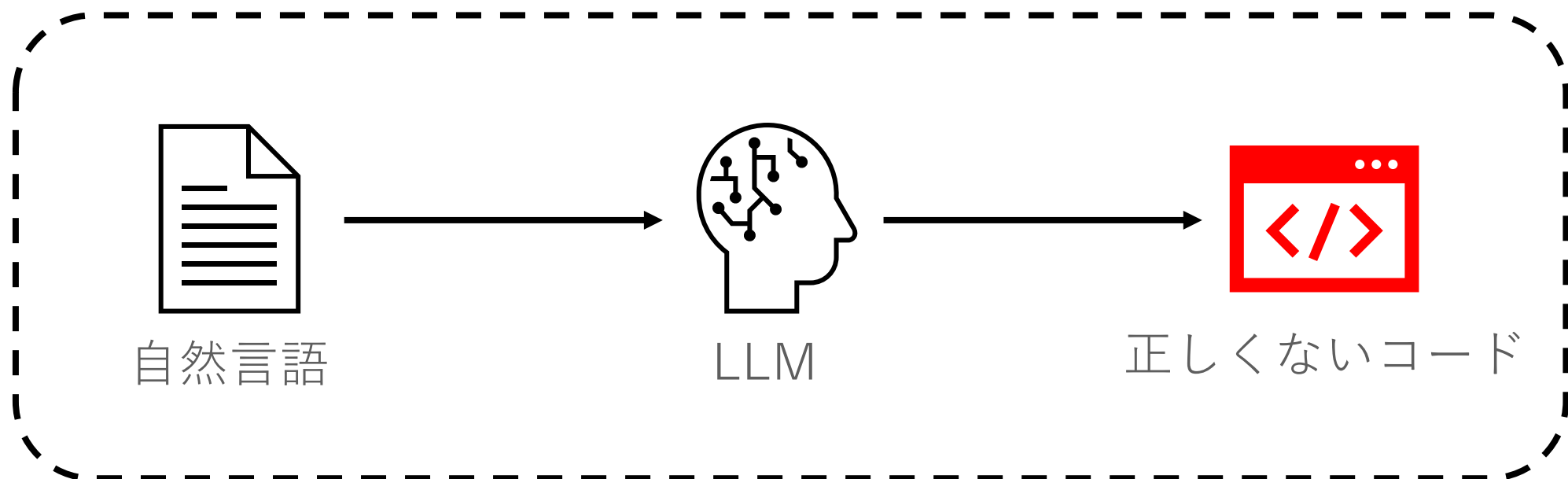
正しいコードの生成



問題点

しかし、LLMは未だに多くのコード生成タスクで正しいコードを生成することができない。

正しくないコードの生成



正しく生成できない原因の調査

先行研究で行われてきた調査

- LLMのAPI誤用の調査[1]
- LLMが入力された要件(≡タスク)をどれだけ明確化できるかの調査[2]
- コード生成タスクにおけるハルシネーションの調査[3]
- Etc...

[1]Zhong, L., & Wang, Z. (2023). Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation. , 21841-21849.

[2]Wu, J., & Fard, F. (2024). HumanEvalComm: Benchmarking the Communication Competence of Code Generation for LLMs and LLM Agent. *ACM Transactions on Software Engineering and Methodology*.

[3]Liu, F., Liu, Y., Shi, L., Huang, H., Wang, R., Yang, Z., & Zhang, L. (2024). Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. *ArXiv*, abs/2404.00971.

正しく生成できない原因の調査

先行研究で行われてきた調査

- LLMのAPI誤用の調査[1]
- LLMが入力された要件(≡タスク)をどれだけ明確化できるかの調査[2]

しかしながら, LLMが出力するエラーに関する理解はまだ限られている。



LLMが出力するエラーに関する調査が求められている

[1]Zhong, L., & Wang, Z. (2023). Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation. , 21841-21849.

[2]Wu, J., & Fard, F. (2024). HumanEvalComm: Benchmarking the Communication Competence of Code Generation for LLMs and LLM Agent. *ACM Transactions on Software Engineering and Methodology*.

[3]Liu, F., Liu, Y., Shi, L., Huang, H., Wang, R., Yang, Z., & Zhang, L. (2024). Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. *ArXiv*, abs/2404.00971.

目的・アプローチ

目的：

- 開発者の既存モデルの限界理解の補助
- 開発者のモデル改善の機会特定の補助

アプローチ

- LLMによるコード生成エラーの詳細な分析
 - 6つの主要なLLMに焦点を当てて分析
- 以下のRQに従って評価を行う
 - RQ1:異なるLLMはどのような種類のコード生成エラーを犯すか？
 - RQ2:コード生成エラーを修正するにはどれくらいの労力が必要か？
 - RQ3:タスクの複雑さはLLMのコード生成にどのように影響するか？
 - RQ4：部分的に失敗したコードは、完全に失敗したコードと比較して異なる特性を示すか？（今回は省略）

実験概要

6つのLLMを用いて実験

- CodeGen-16B, InCoder-1.3B, GPT-3.5, GPT-4, SantaCoder, およびStarCoder
- TABLE 1から分かる通り, 幅広いモデルサイズと性能のLLMが用いられている

TABLE I: Code generation LLMs used in this study

Model	Release	Size	Performance	
			<i>Pass@1</i>	<i>Incorrect Solutions</i>
CodeGen-16B [1]	Mar. 2022	16B	32.9%	110
InCoder-1.3B [2]	Apr. 2022	1.3B	12.2%	144
GPT-3.5 [5]	Nov. 2022	175B	73.2%	42
GPT-4 [4]	Mar. 2023	1.7T	89.0%	18
SantaCoder [6]	Apr. 2023	1.1B	14.6%	139
StarCoder [3]	May. 2023	15.5B	34.1%	104

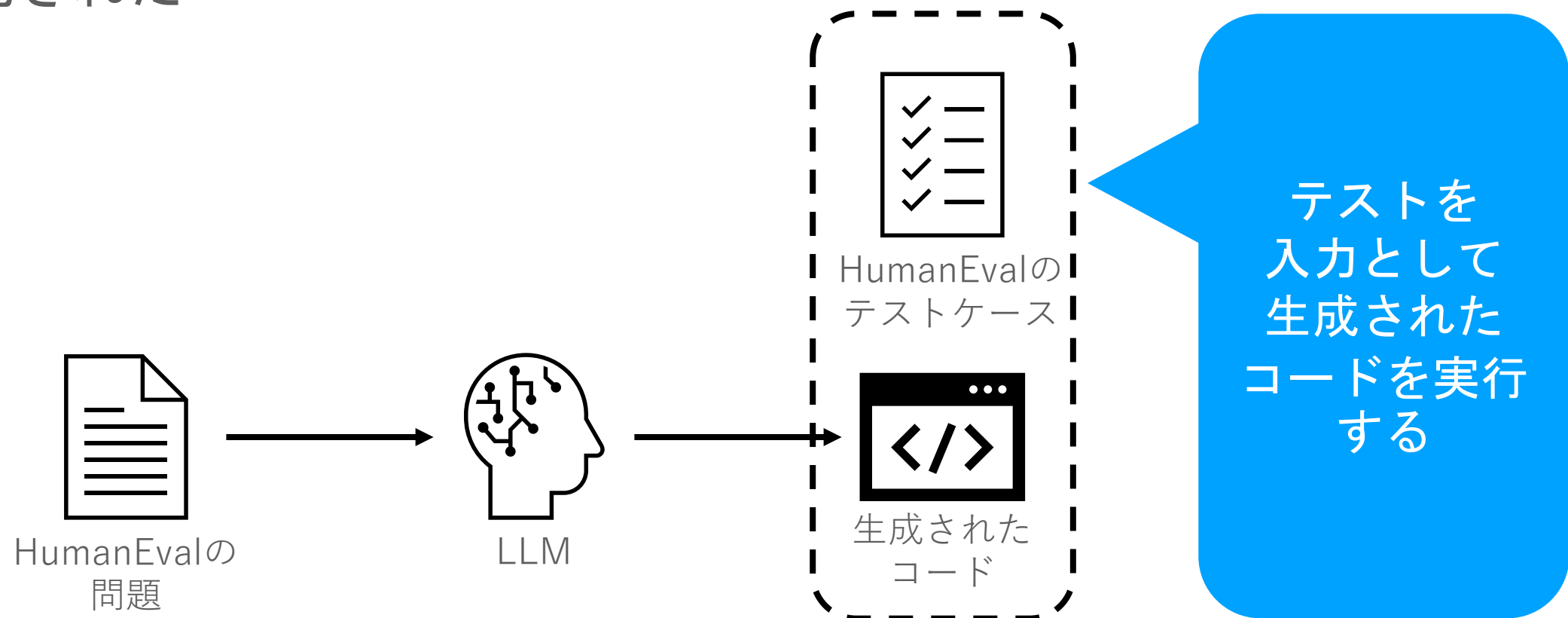
実験概要

6つのLLMにHumanEvalを適用する

- LLMのtemperatureは0に設定

以下の条件を満たした場合，該当コードはエラーを含むものとする

- HumanEvalのテストを通過しなかった
- HumanEvalのテストに通過したが，目視で確認した際にエラーが発見された



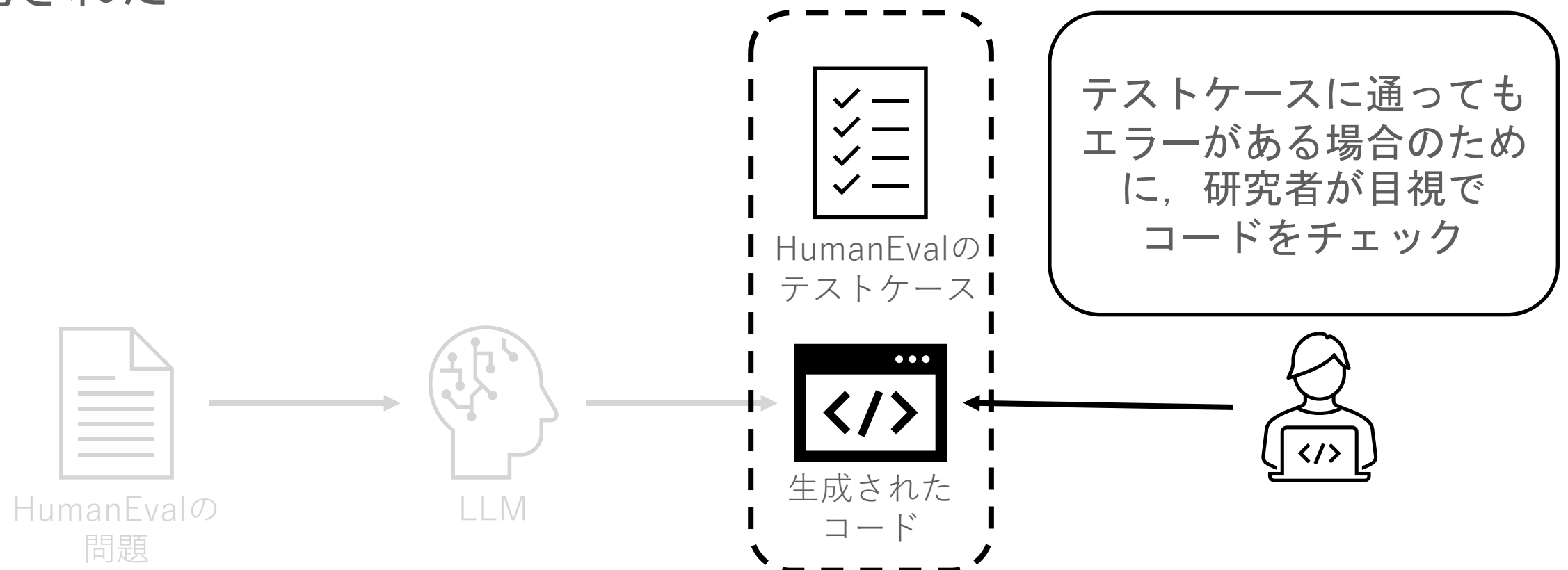
実験概要

6つのLLMにHumanEvalを適用する

- LLMのtemperatureは0に設定

以下の条件を満たした場合，該当コードはエラーを含むものとする

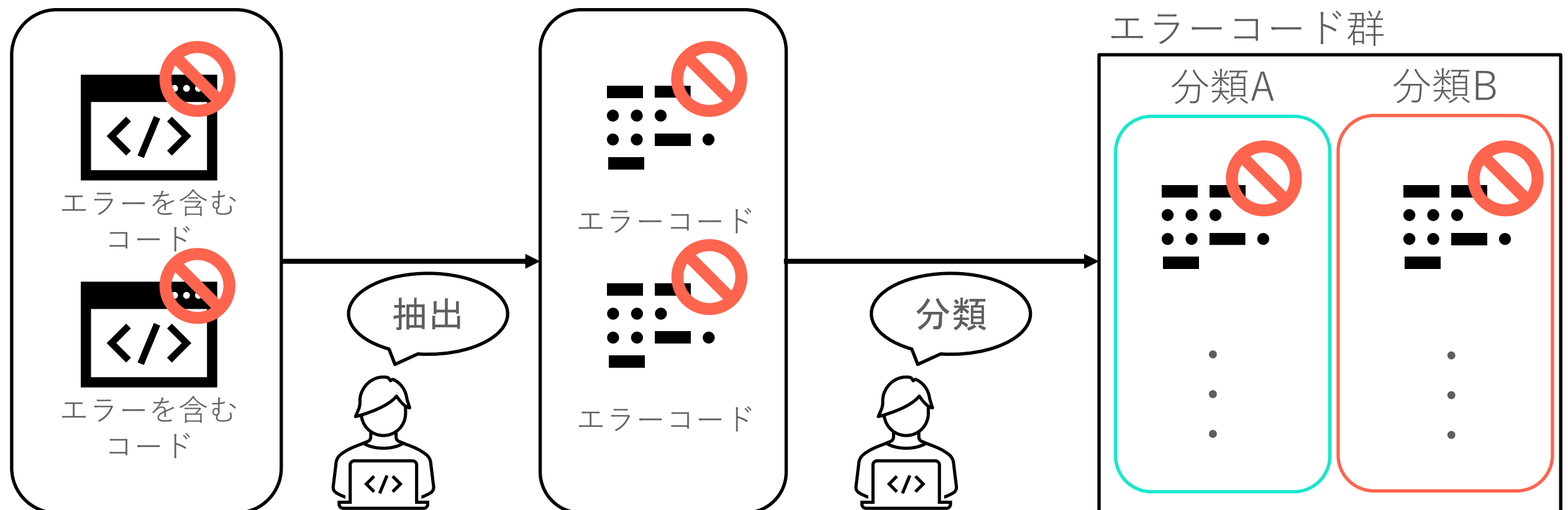
- HumanEvalのテストを通過しなかった
- HumanEvalのテストに通過したが，目視で確認した際にエラーが発見された



実験概要

オープンコーディングによりコード生成エラーの分類を行った。

- 抽出, 分類共に著者らが人手で行っている。
 - 抽出のために, 人手でステップバイステップのデバッグを行っている



結果：異なるLLMはどのような種類のコード生成エラーを犯すか？

この研究では、発見したエラーをおおまかに2つに分類している

- 意味的エラー

- Ex. 条件の欠落, 意図と異なるロジック等
- このエラーの理解により, LLMがどの程度タスクを正しく解釈できているか, 意味的に正しいプログラムを生成できるかが理解できる

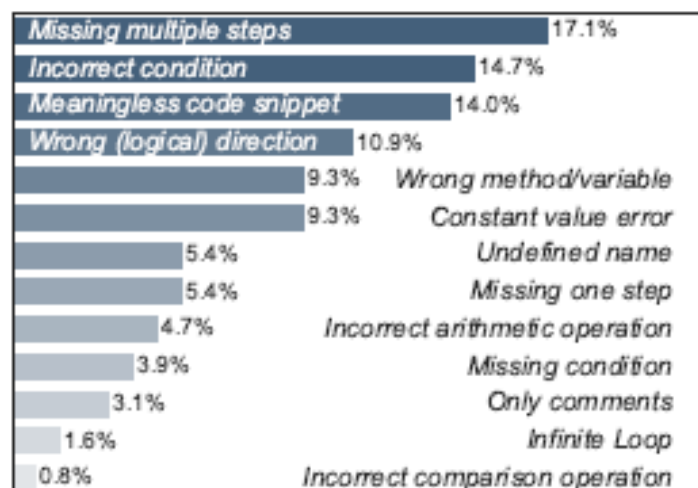
- 構文エラー

- Ex. 正しくないコードブロック, 正しくない引数等
- このエラーの理解により,
 - LLMのコード構造生成能力をよりよく評価できる.
 - LLMによるコード生成エラーを特定し, 修復するための新しい手法の設計に役立つ.

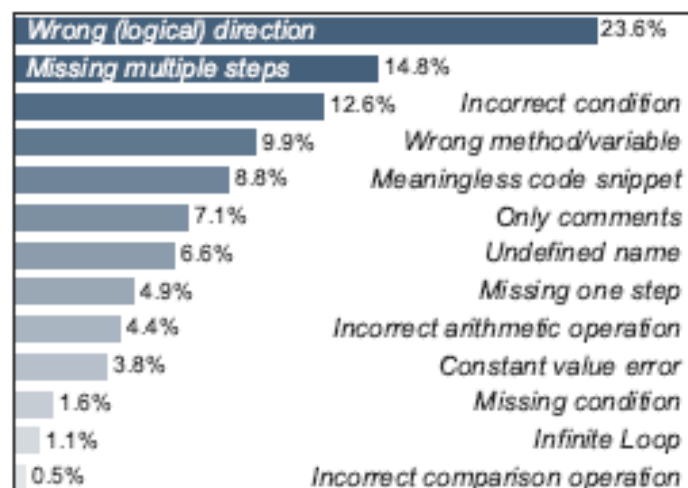
細かい分類は補足資料に掲載

結果：異なるLLMはどのような種類のコード生成エラーを犯すか？

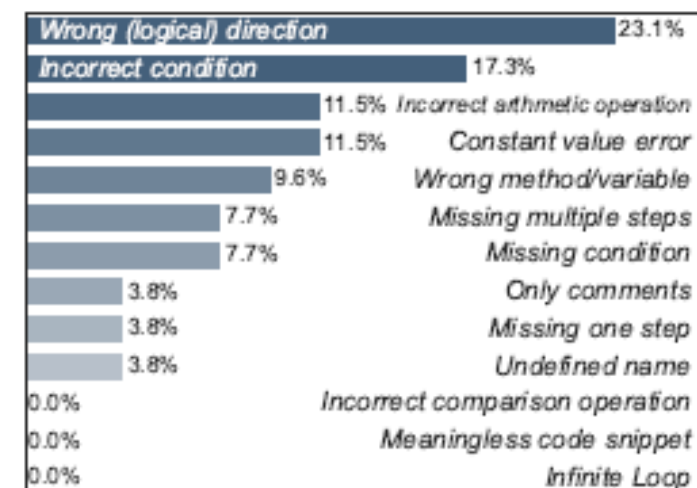
以下に示す図は，6つのLLMそれぞれで発生した意味的エラーの割合である．



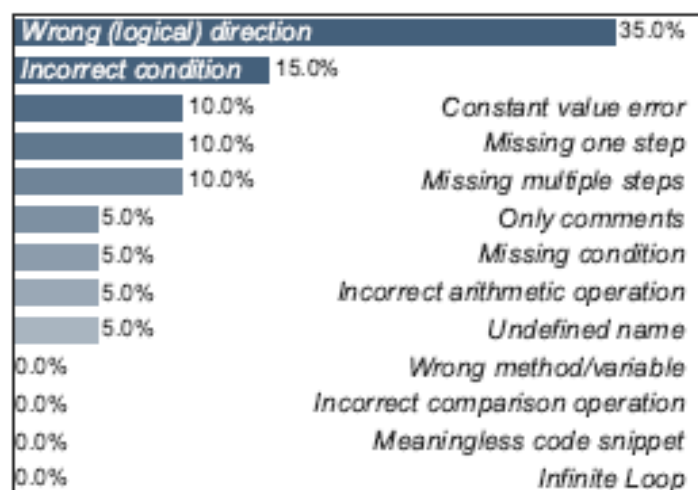
(a) CodeGen-16B



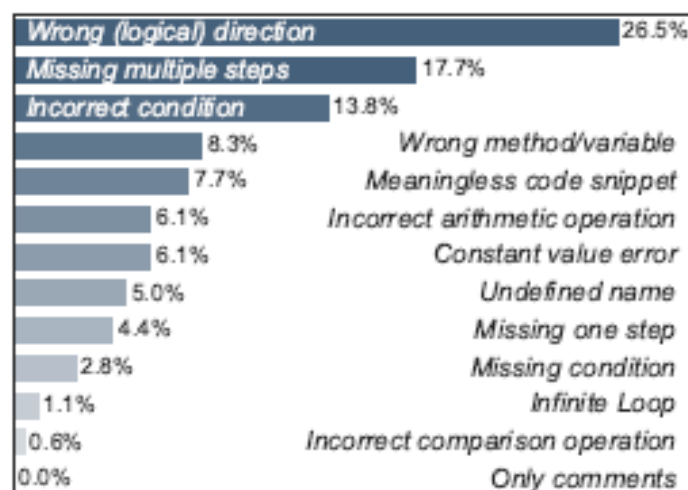
(b) InCoder-1.3B



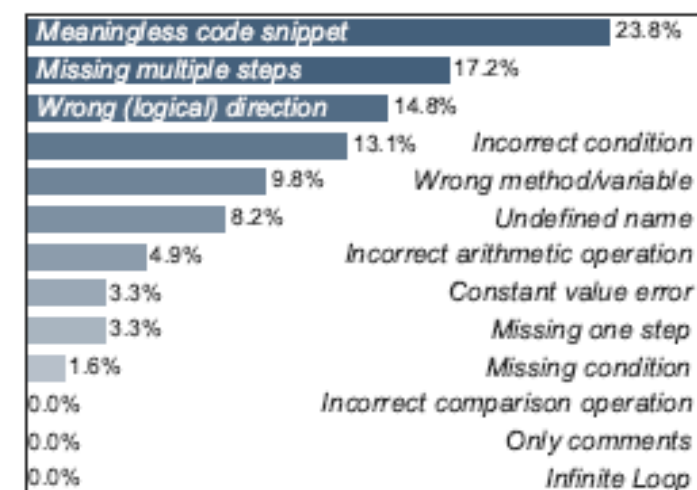
(c) GPT-3.5



(d) GPT-4



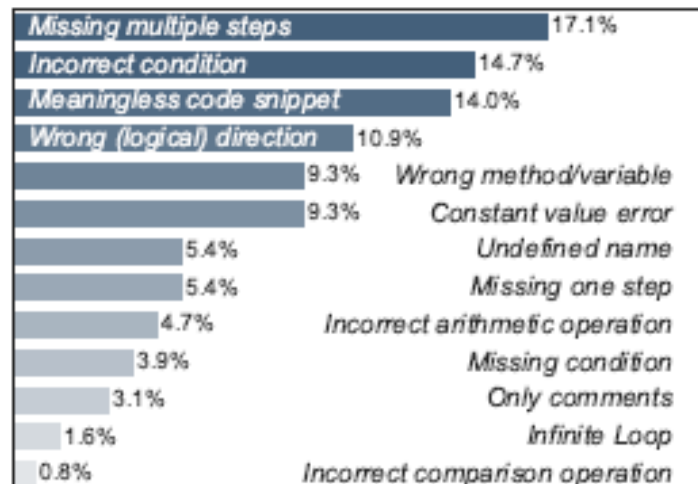
(e) SantaCoder



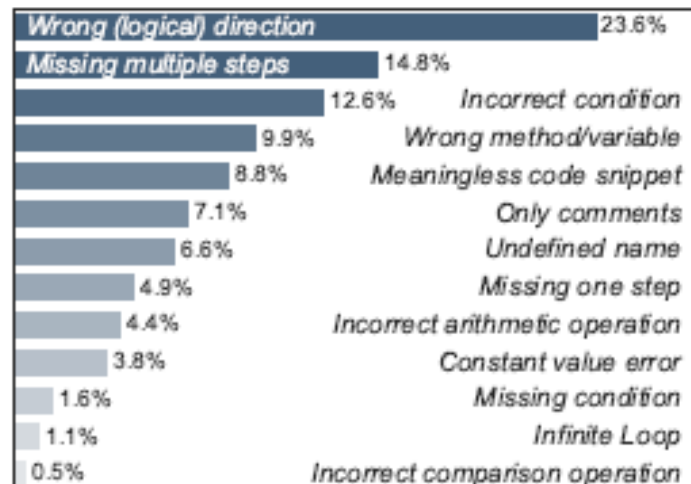
(f) StarCoder

結果：異なるLLMはどのような種類のコード生成エラーを犯すか？

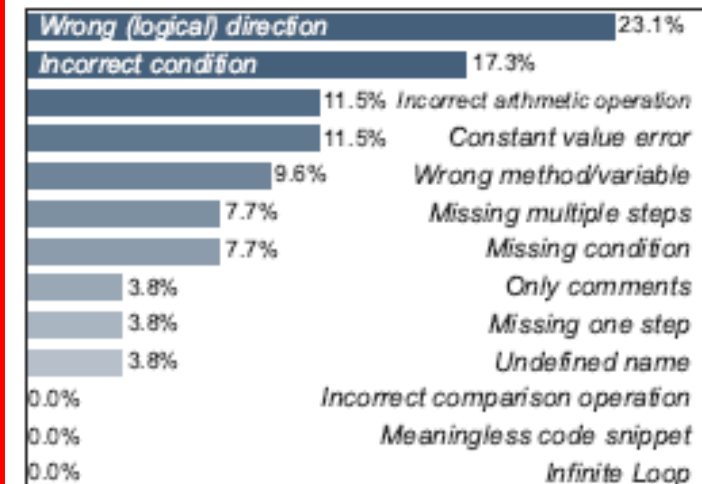
Code-Gen-16Bのような**小さいサイズのモデル**は、
意味のないコードや複数のステップが欠落したコードを生成しやすい。
GPT-4のような**大きいサイズのモデル**は、
定数値エラーや算術演算エラーを犯しやすい。



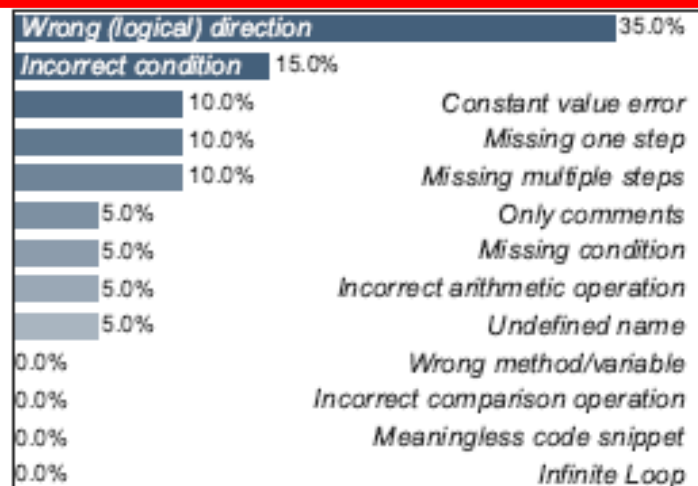
(a) CodeGen-16B



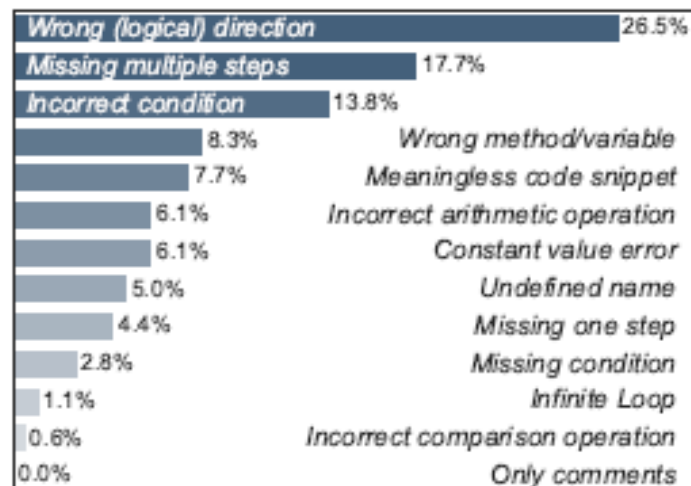
(b) InCoder-1.3B



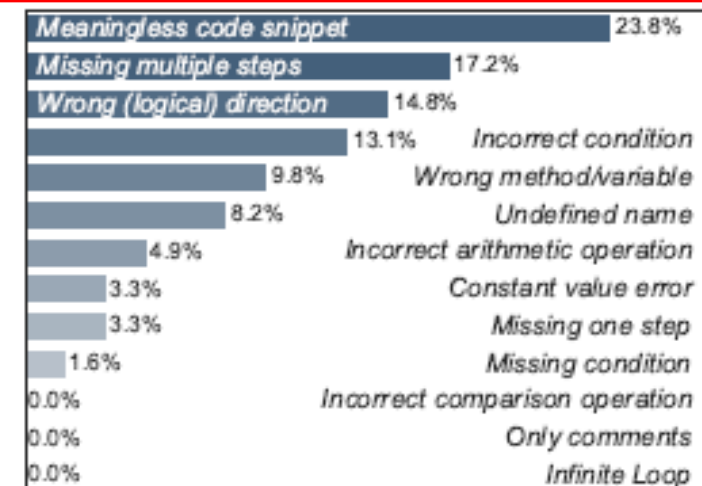
(c) GPT-3.5



(d) GPT-4



(e) SantaCoder



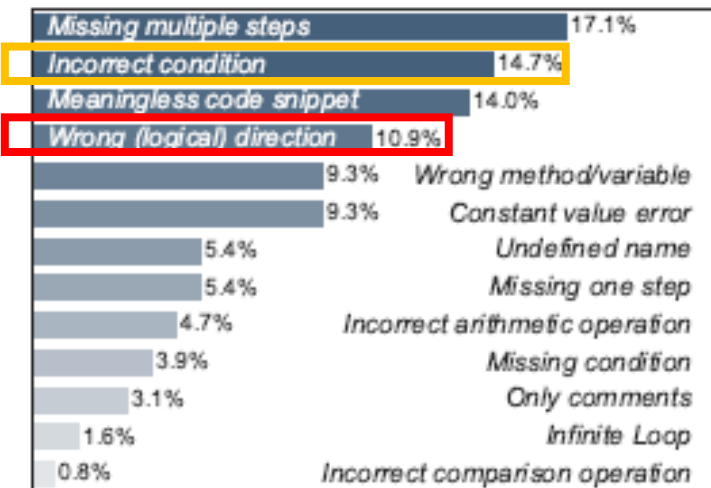
(f) StarCoder

結果：異なるLLMはどのような種類のコード生成エラーを犯すか？

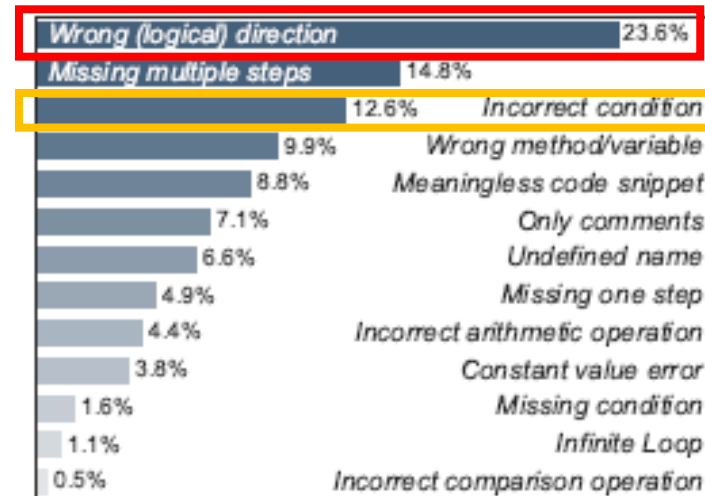
Incorrected cordition(正しくない条件)や, **Wrong direction**(意図と異なるロジック)等いくつかのエラーが, すべてのLLMで共通して上位に入っている



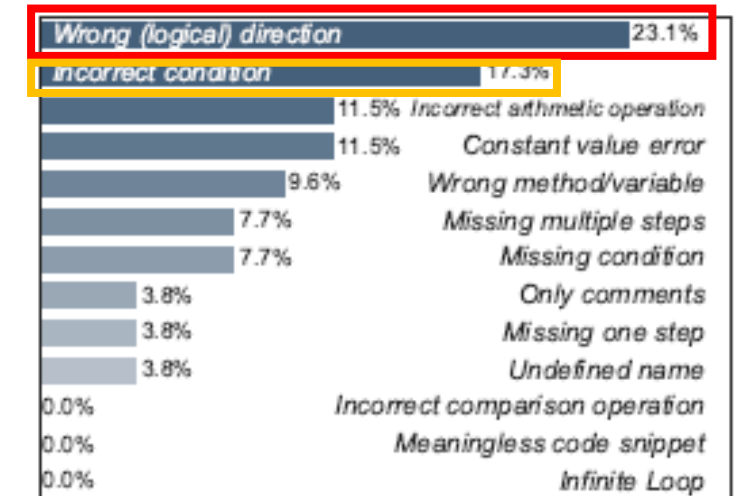
いくつかのエラーは, LLMのサイズ, 性能を問わず共通して発生しやすい



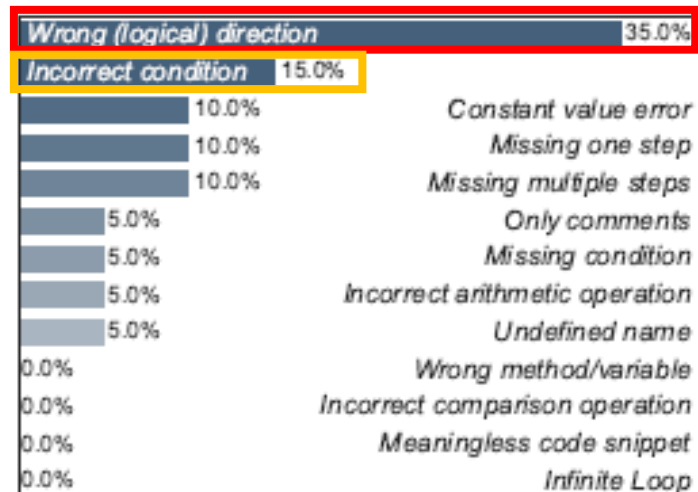
(a) CodeGen-16B



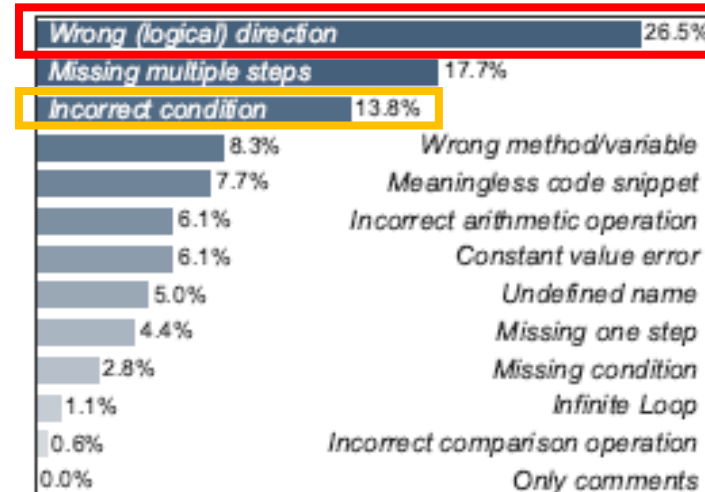
(b) InCoder-1.3B



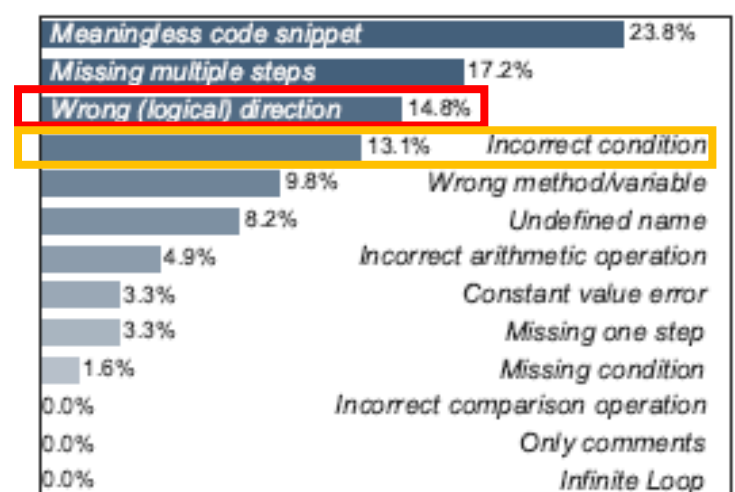
(c) GPT-3.5



(d) GPT-4



(e) SantaCoder



(f) StarCoder

結果：異なるLLMはどのような種類のコード生成エラーを犯すか？

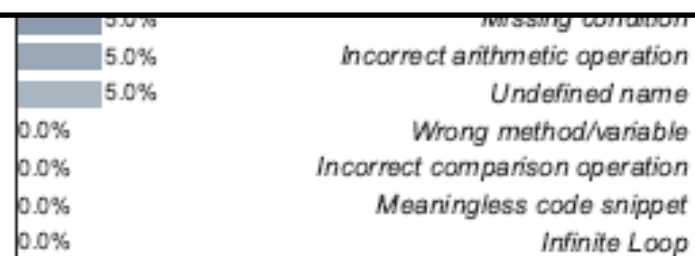
Incorrected condition(正しくない条件)や, Wrong direction(意図と異なるロジック)等いくつかのエラーが, すべてのLLMで共通して上位に入っている



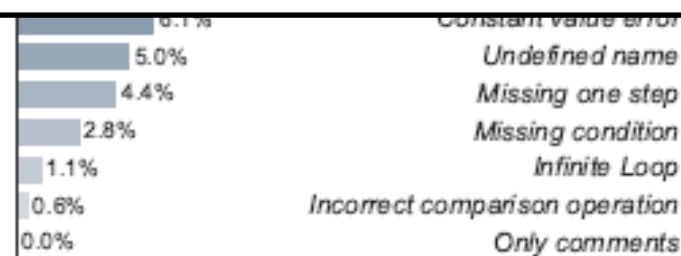
発見 1 :

6つのLLM間で最も一般的な意味的エラーは, **意図しないロジック**と**正しくない条件**である。これは, すべてのLLMが複雑なタスク要件を解釈し, 正しい論理条件を生成するのに苦労していることを示している。

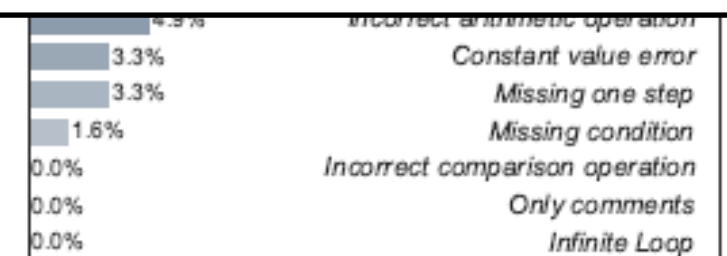
また, GPT-4やGPT-3.5のような超大規模モデルと比較して, 小さなモデルは, より多くの意味のないコードと複数のステップが欠落したコードを生成する。



(d) GPT-4



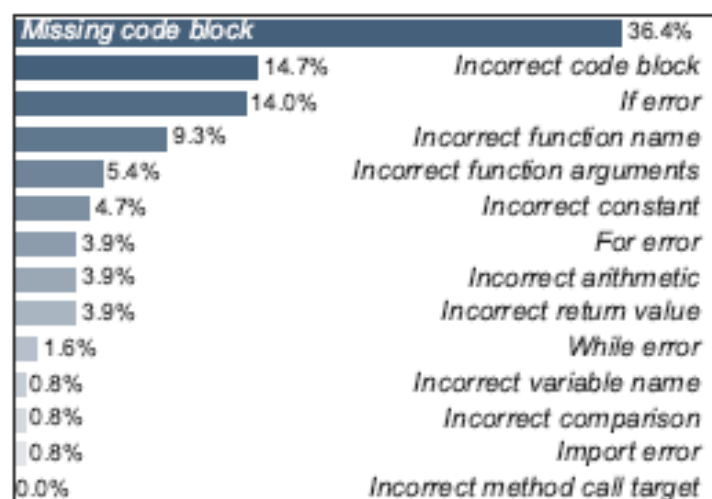
(e) SantaCoder



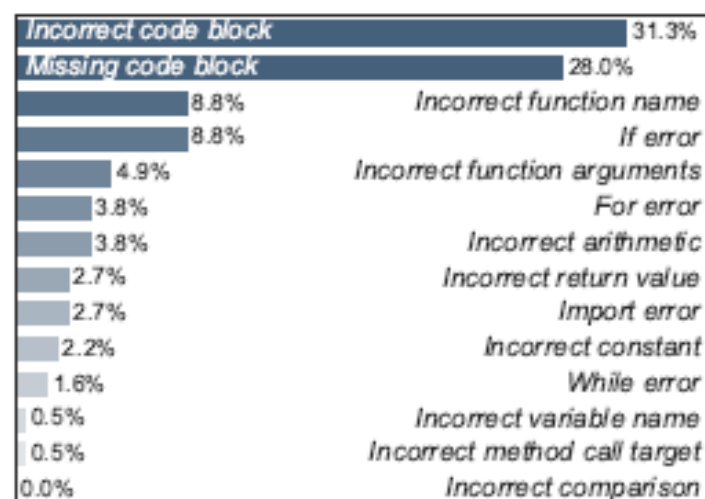
(f) StarCoder

結果：異なるLLMはどのような種類のコード生成エラーを犯すか？

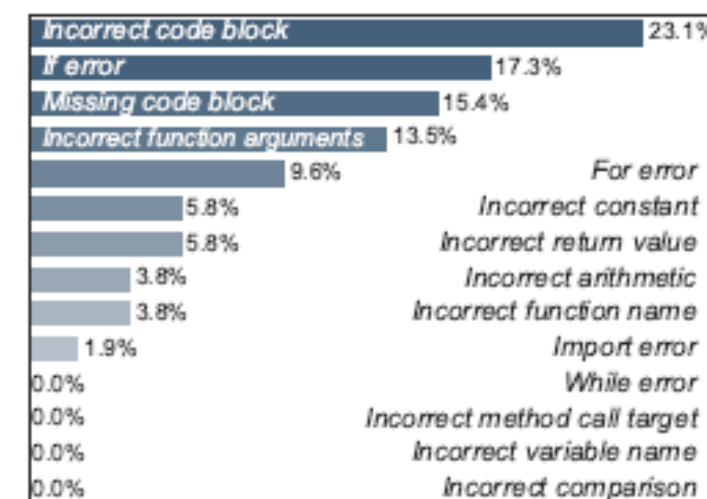
以下に示す図は，6つのLLMそれぞれで発生した構文エラーの割合である．



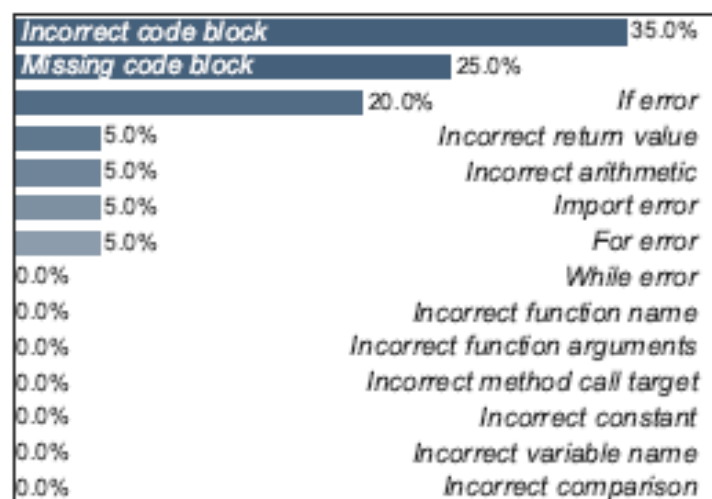
(a) CodeGen-16B



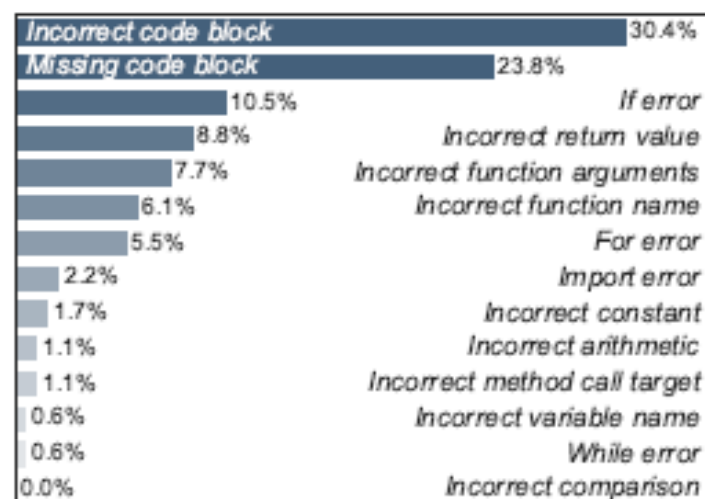
(b) InCoder-1.3B



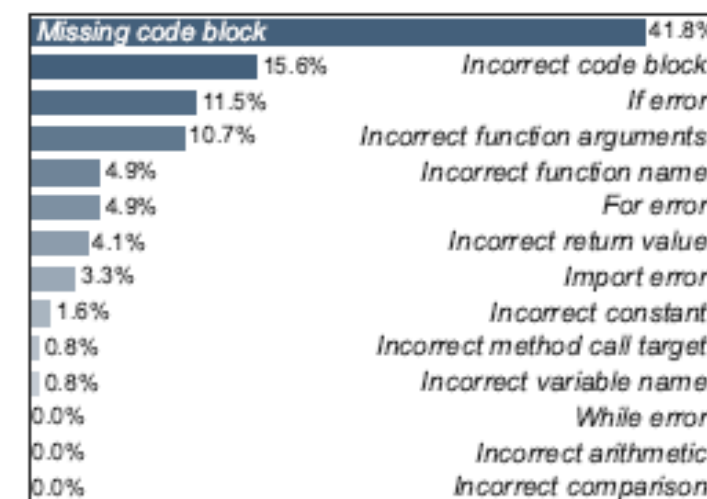
(c) GPT-3.5



(d) GPT-4



(e) SantaCoder



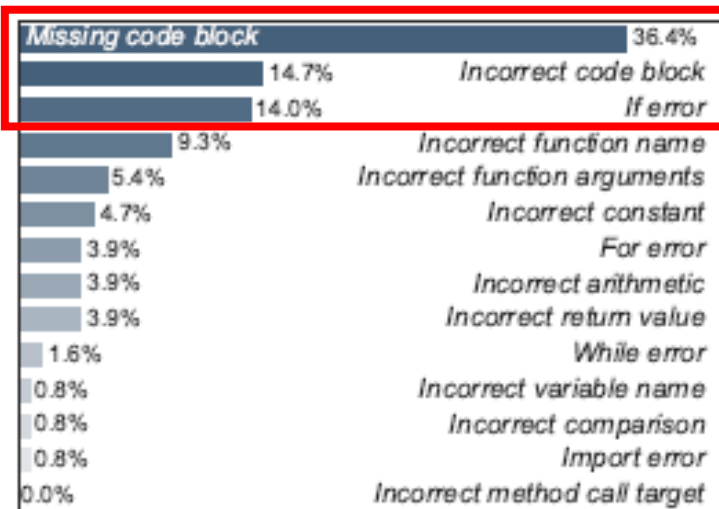
(f) StarCoder

結果：異なるLLMはどのような種類のコード生成エラーを犯すか？

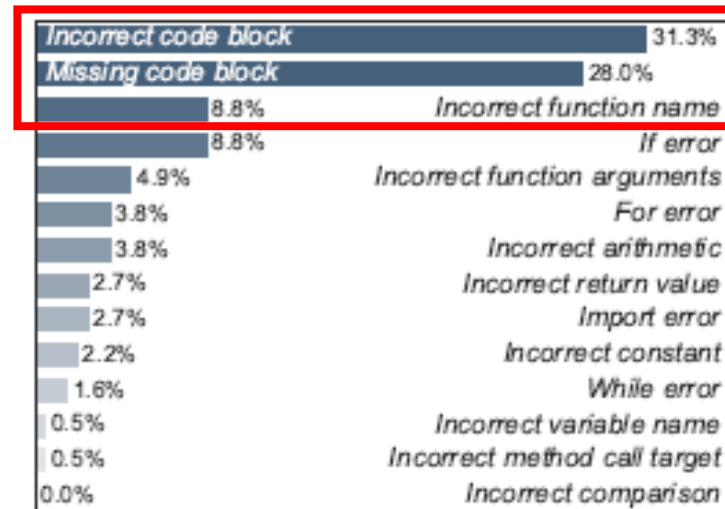
すべてのLLMの上位3つのエラーは、コードブロック、またはif文のいずれかで発生している。



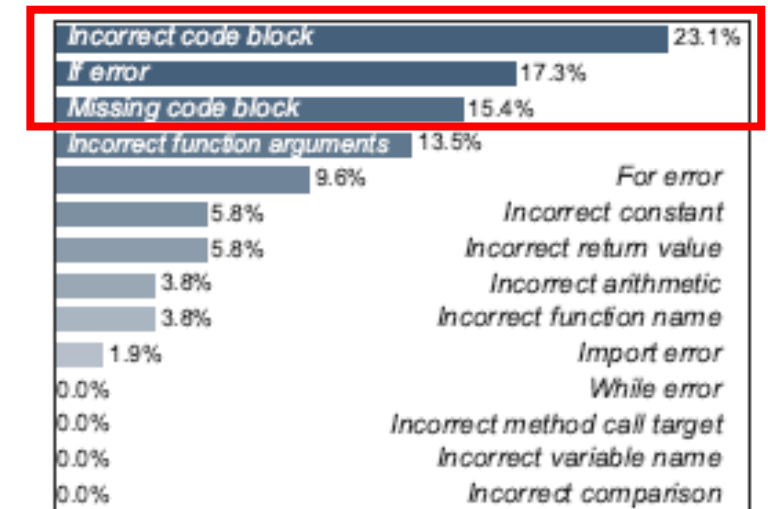
LLMはそのサイズや性能に関わらず、コードブロック全体を生成するのは難しい



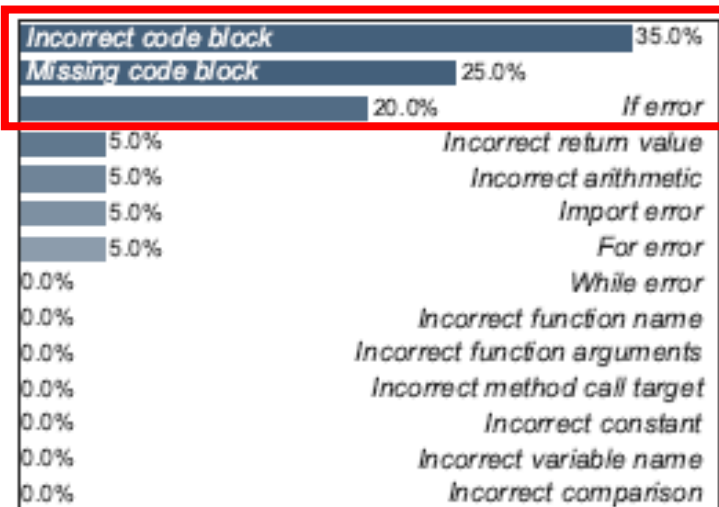
(a) CodeGen-16B



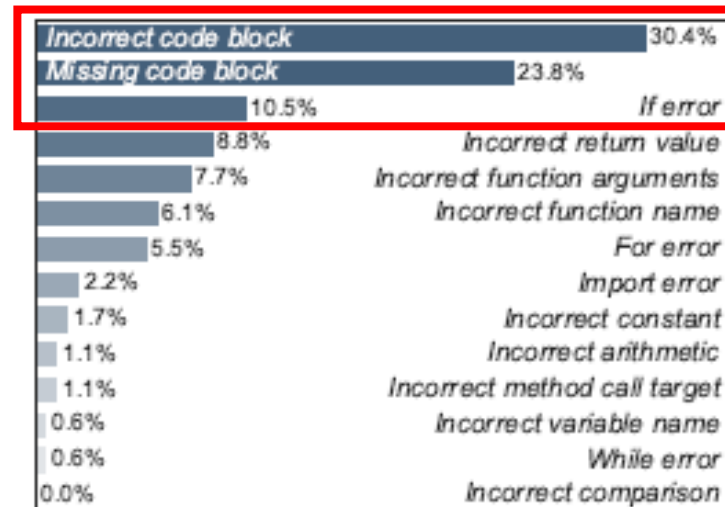
(b) InCoder-1.3B



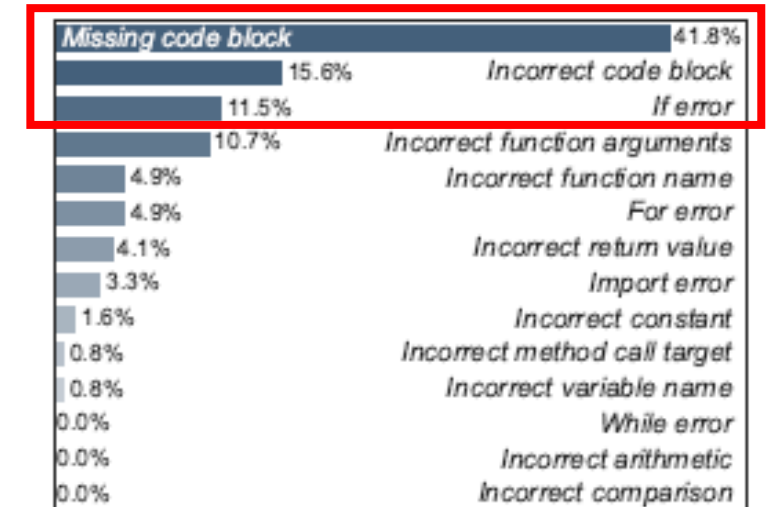
(c) GPT-3.5



(d) GPT-4



(e) SantaCoder



(f) StarCoder

結果：異なるLLMはどのような種類のコード生成エラーを犯すか？

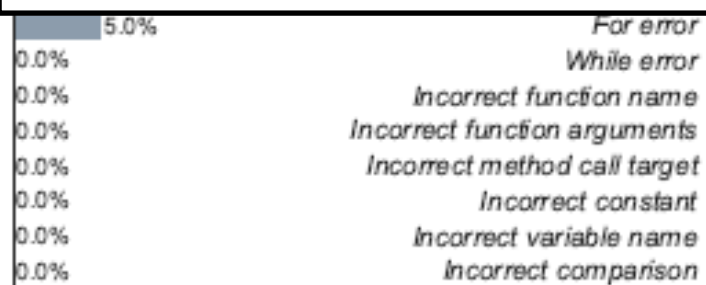
すべてのLLMの上位3つのエラーは、コードブロック、またはif文のいずれかで発生している。



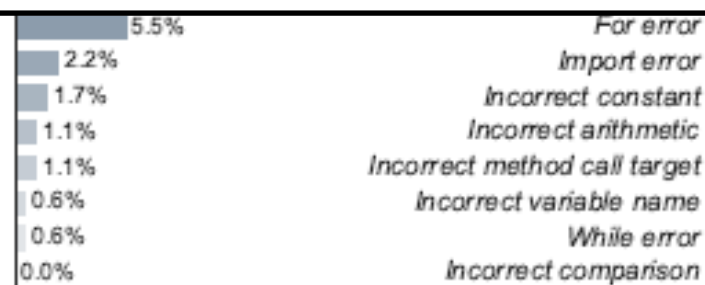
LLMはそのサイズや性能に関わらず、コードブロック全体を生成するのは難しい

発見2：

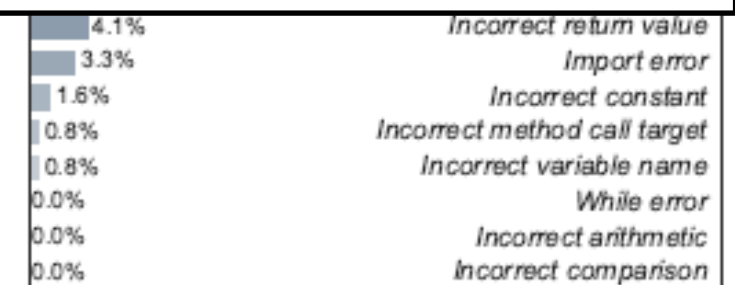
6つのLLMによって作成された構文的エラーの40%以上は、欠落/不正確なコードブロックである。



(d) GPT-4



(e) SantaCoder



(f) StarCoder

結果：コード生成エラーの修復労力

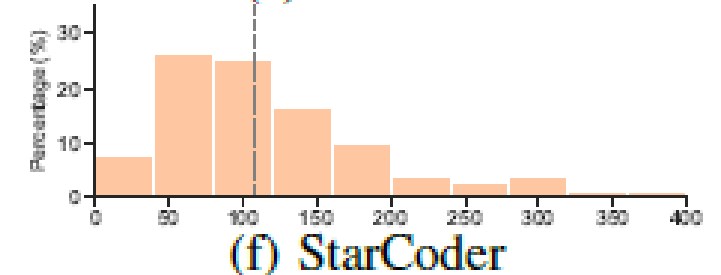
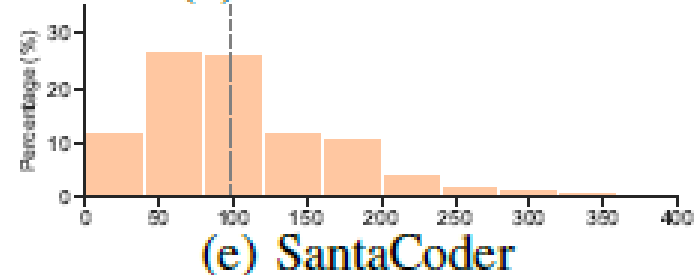
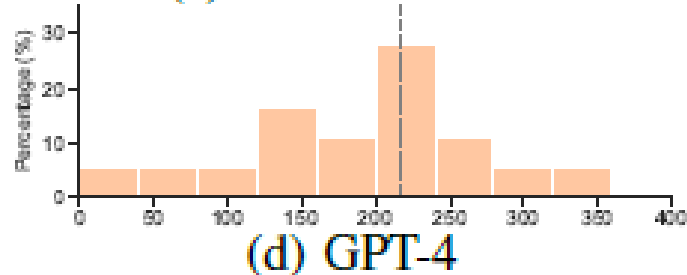
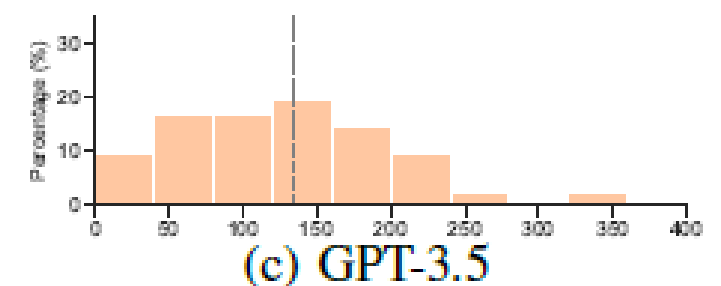
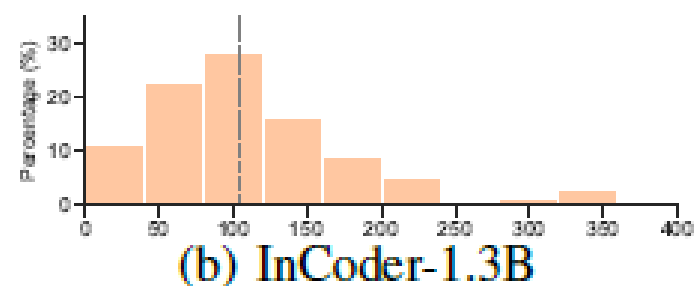
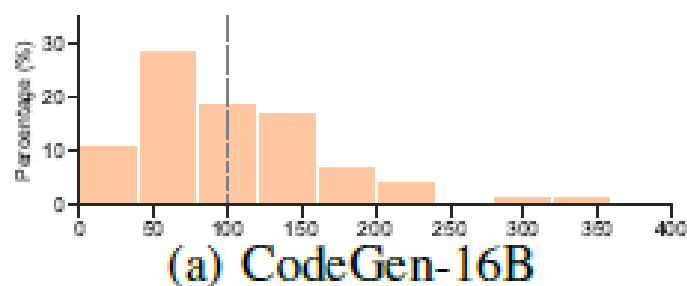
正しいコードと、エラーを含むコードを比較することでコード生成エラーの修復労力を測る

- ここでいう正しいコードは、HumanEvalのグランドトゥールース
- メトリックとして、レーベンシュタイン距離、Jaccard係数、CodeBERTScoreを使用

結果：コード生成エラーの修復労力

以下の図は各LLMのレーベンシュタイン距離の分布を示している

- 縦軸はすべての生成エラーに対する各数値帯の割合
- 横軸はレーベンシュタイン距離

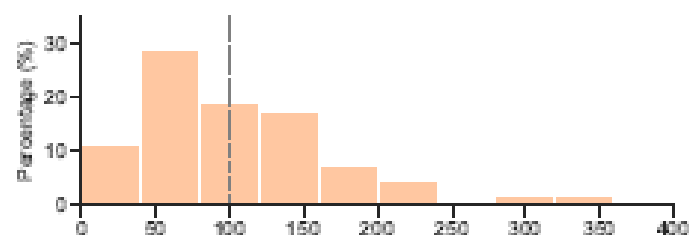


結果：コード生成エラーの修復労力

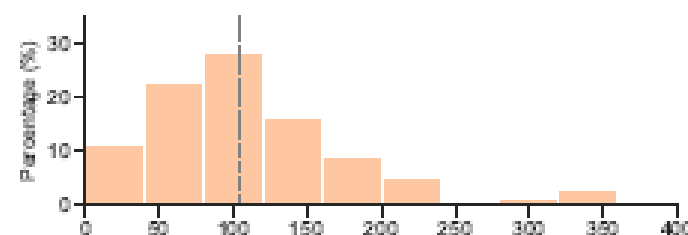
各LLMのレーベンシュタイン距離は共通して

- 中央値が100前後となっている
- 生成されたコードの80%以上に関して、距離が50を越えている

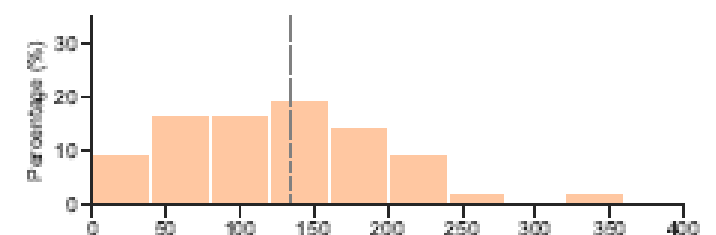
(Jaccard係数に関して同様な傾向が得られた)



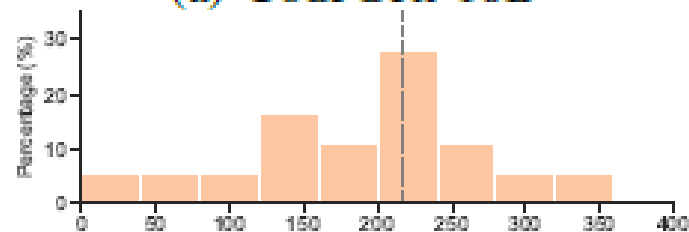
(a) CodeGen-16B



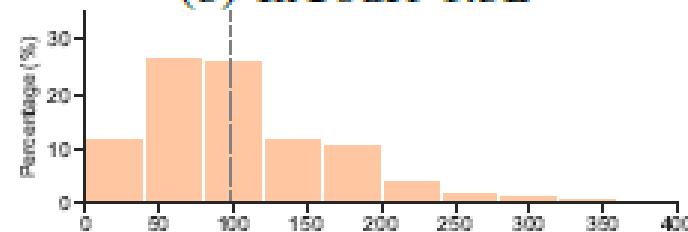
(b) InCoder-1.3B



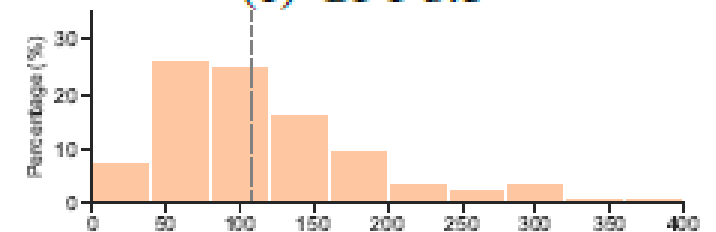
(c) GPT-3.5



(d) GPT-4



(e) SantaCoder

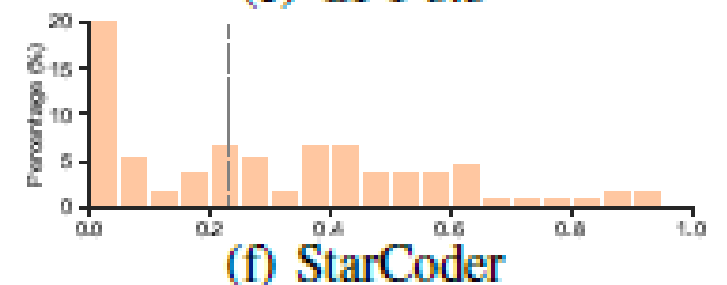
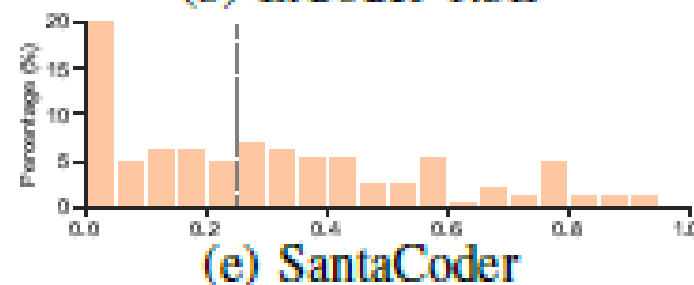
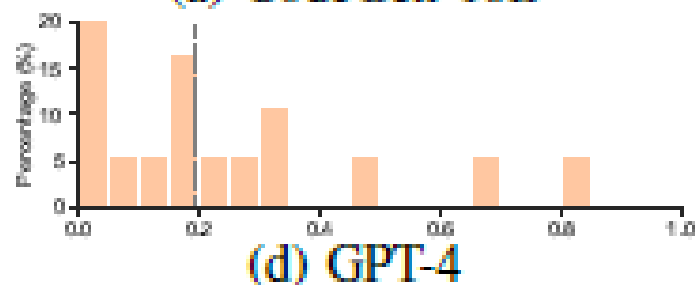
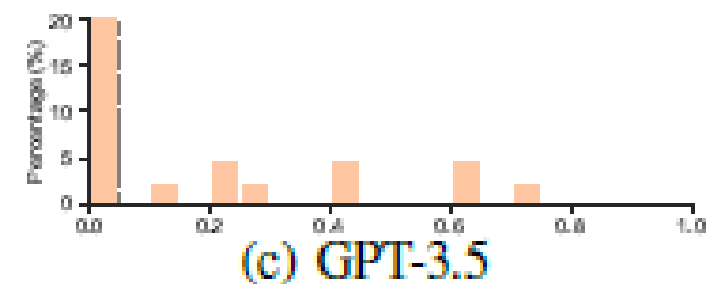
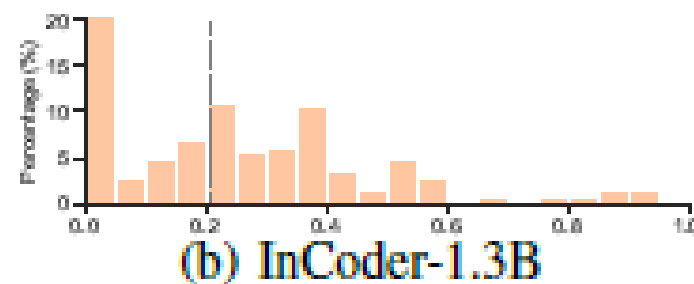
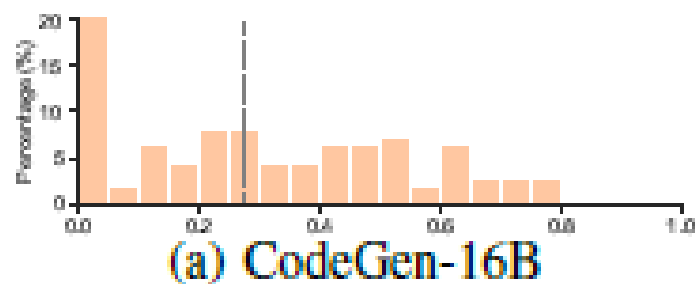


(f) StarCoder

結果：コード生成エラーの修復労力

以下の図は各LLMのCodeBERTScoreの分布を示している

- 縦軸はすべての生成エラーに対する各数値帯の割合
- 横軸はCodeBERTScore

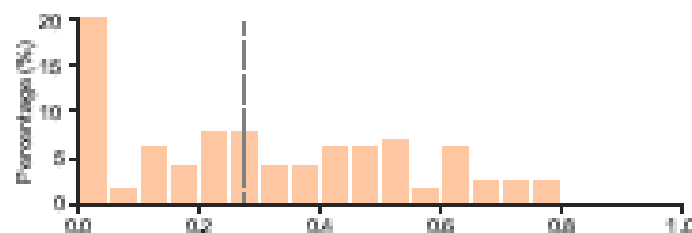


結果：コード生成エラーの修復労力

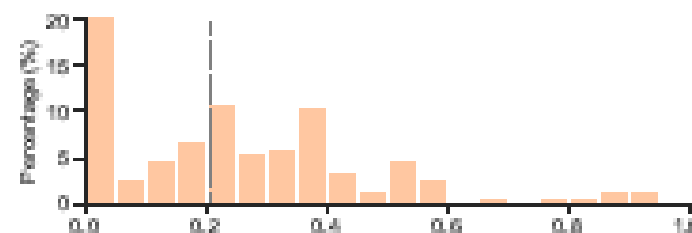
- 中央値がGPT-3.5を除いて0.2程度(GPT-3.5のみ0.05)
- すべてのLLMにおいて、CodeBERTScoreが0であるものが30%以上存在している



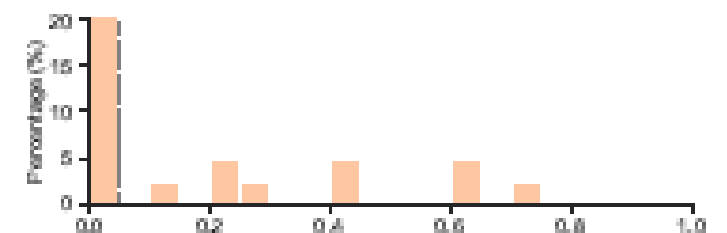
LLMが出力するエラーの多くは、意味的にも正しいコードと大きく異なる



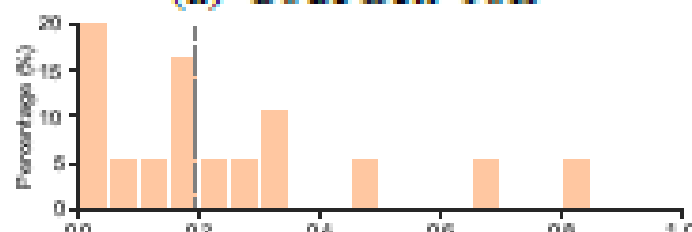
(a) CodeGen-16B



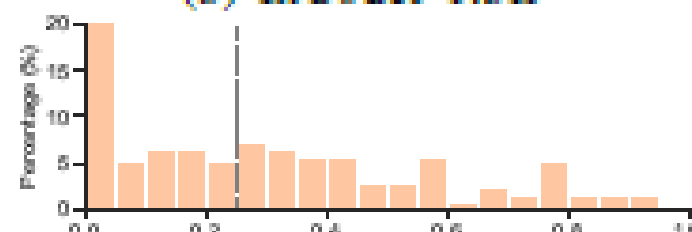
(b) InCoder-1.3B



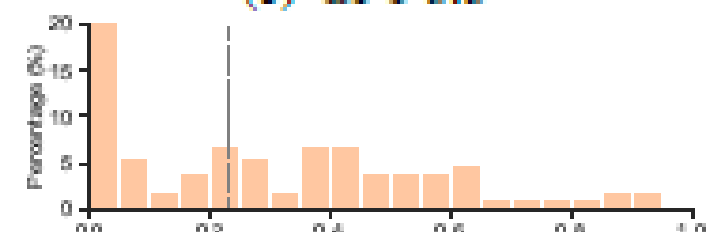
(c) GPT-3.5



(d) GPT-4



(e) SantaCoder



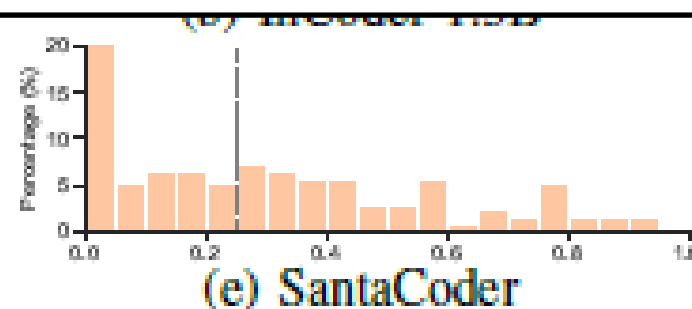
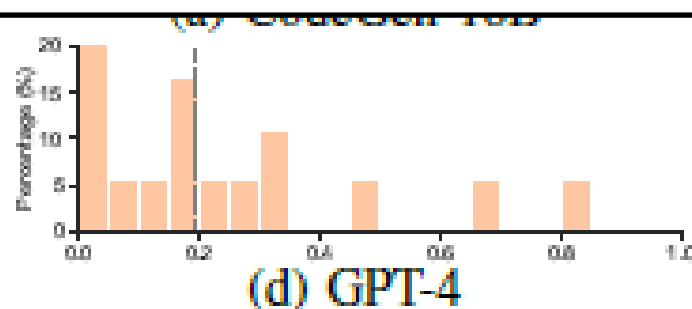
(f) StarCoder

結果：コード生成エラーの修復労力

- 中央値がGPT-3.5を除いて0.2程度(GPT-3.5のみ0.05)
- すべてのLLMにおいて、CodeBERTScoreが0であるもの

発見3：

6つのLLMによって生成された誤ったコードの大部分は、正しいコードから大きく逸脱している。これは、LLMが生成したコードの修正には、たくさんの労力が必要であることを示唆している。



結果：タスクの複雑さの影響

タスク(=プロンプト)の複雑さが，HumanEvalのテスト通過の可否に与える影響を見る

- しかし，既存研究でこれを評価する指標は提案されていない

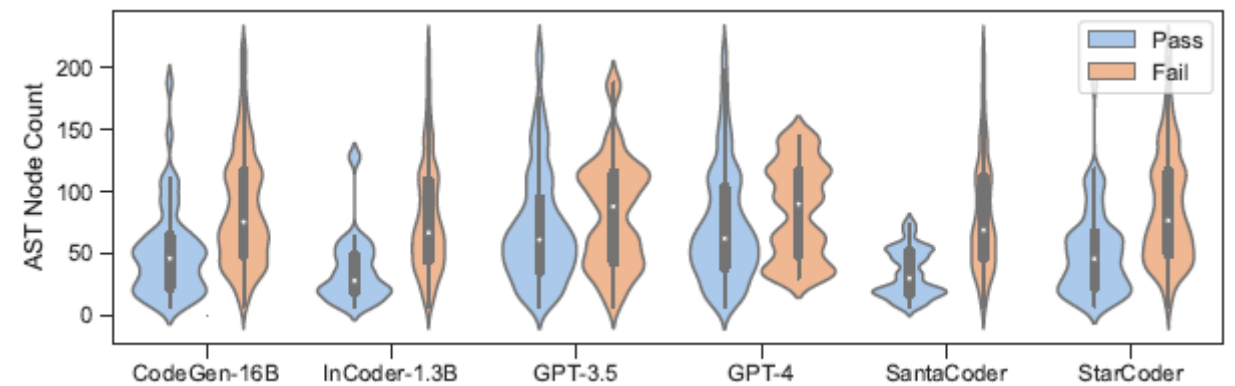
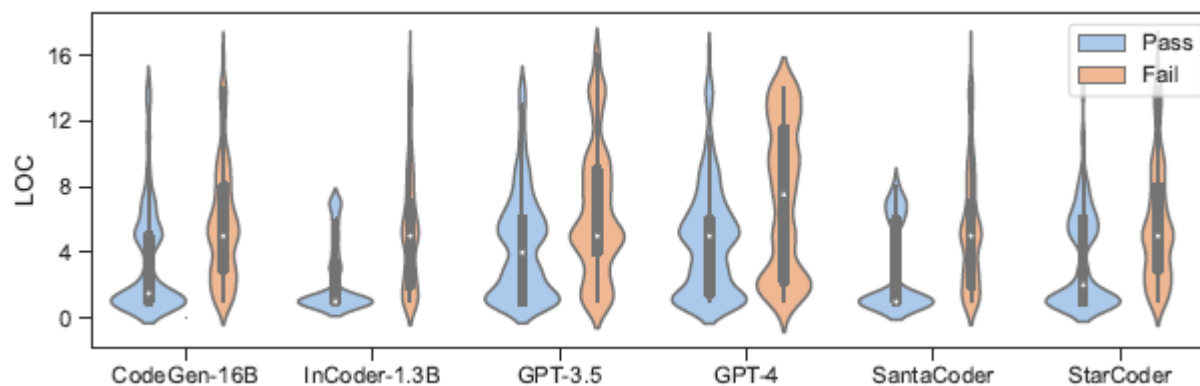
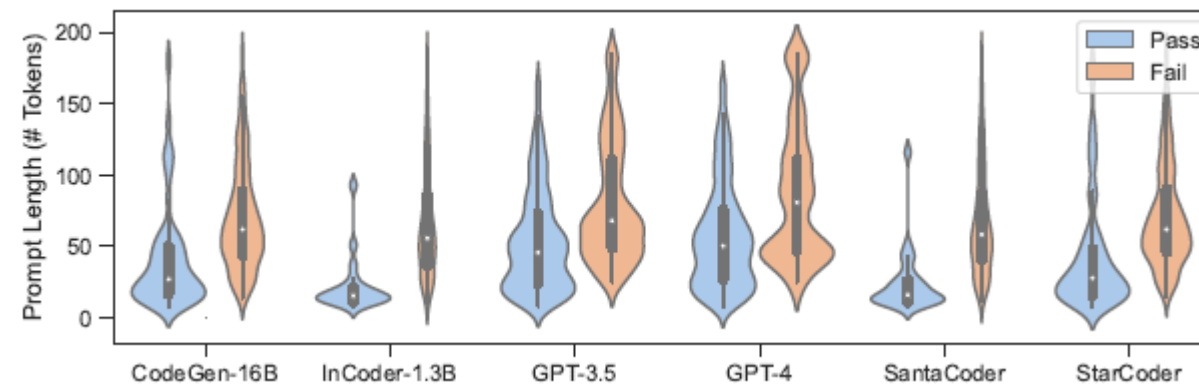
プロンプトの複雑さを測る指標として以下2つを検討

- プロンプトの単語数
- HumanEvalの正解コードの行数
 - しかし，行によって表現の複雑さが異なるため，一行の価値が同じとは限らないとも考えられる
 - ex. `x=1+1`と`y=func(x)*sum([x+1 in [1,2,3]])`だと明らかに表現の複雑さが異なる
 - そのため，この指標の妥当性の検証のために，HumanEvalの正解のASTノードの数を利用している

HumanEvalのテスト成功/失敗における各指標の有意差を調べるために，マン・ホイットニーのU検定を行った

結果：タスクの複雑さの影響

以下の図は、6つのLLMに対するプロンプトの単語数、正解コード行数、正解ASTノード数の分布をHumanEvalのテスト成功/失敗に分けて示したバイオリン図である

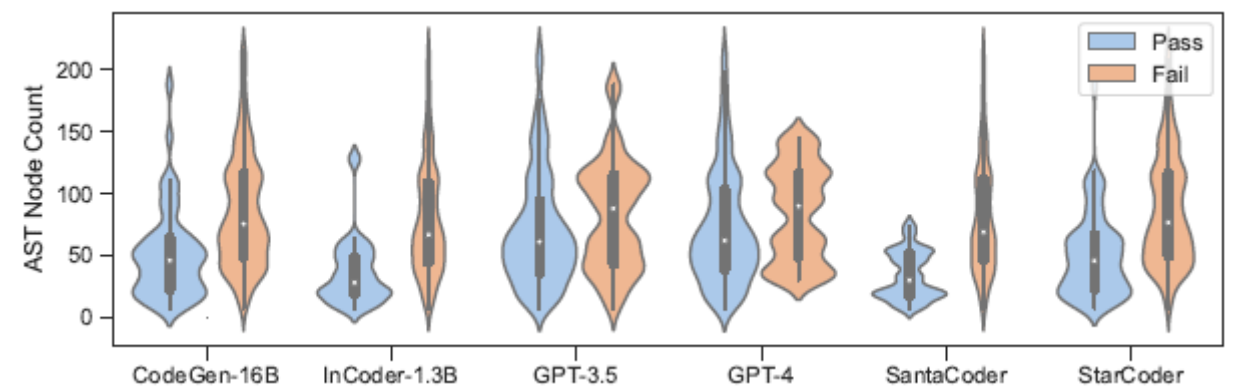
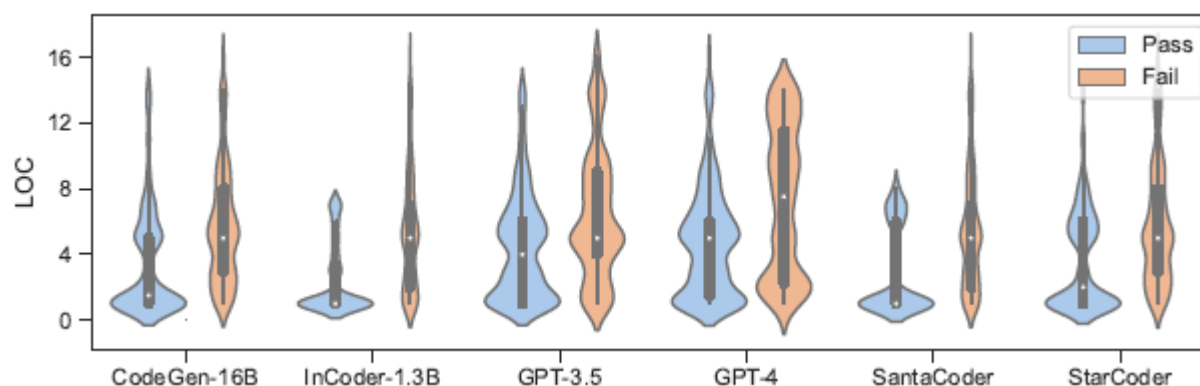
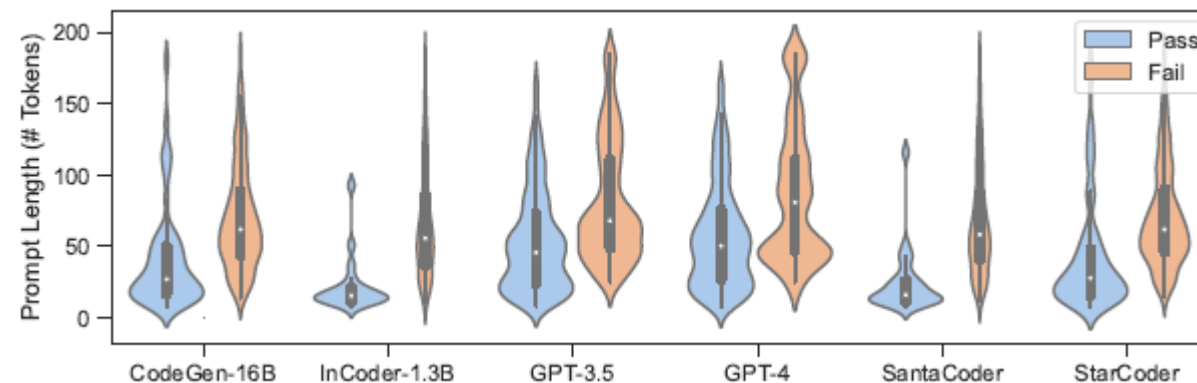


結果：タスクの複雑さの影響

6つのLLMにおけるプロンプトの単語数,
正解コード行数, 正解ASTノード数の三指標で,
テストの可否の間で有意差が観測された.



タスクの複雑さはテスト通過の可否に影響を与える



結果：タスクの複雑さの影響

失敗したタスクの中でも、プロンプトの単語数が150を越える複雑なタスクにおいて、ある特徴がみられた。

- この失敗したタスクで見られたエラーの大部分(64%)が, **garbage code(役割のないコードスニペット)**であった
- 以下の図はプロンプトと、それに対してInCoder-1.3Bが出力したgarbage codeを含むコードである
 - CodeGen-16Bでも同様な現象が見られた

```
# [Task 129] Given a grid with N rows and N columns (N >=
# 2) and a positive k, each cell of ... (230 words left)
def minPath(grid, k):
    ans.visited = [], {}
    visited[grid[0][0]] = True
    ans.append(grid[0][0])
    ans.append(grid[0][1])
    ...
```

結果：タスクの複雑さの影響

失敗したタスクの中でも、プロンプトの単語数が150を越える複雑なタスクにおいて、ある特徴がみられた。

- この失敗したタスクで見られたエラーの大部分(64%)が、garbage code(役割のないコードスニペット)であった
- 以下の図はプロンプトと、それに対してInCoder-1.3Bが出力したgarbage codeを含むコードである
 - CodeGen-16Bでも同様な現象が見られた

```
# 与えられた2次元配列内で、そのパスに沿った値の合計が最小になるような、長さkのパスを返し  
# てください(動的計画法を用いるのが正解)
```

```
def minPath(grid, k):  
    ans.visited = [], {}  
    visited[grid[0][0]] = True  
    ans.append(grid[0][0])  
    ans.append(grid[0][1])  
    ...
```

回答に関係ない操作を行っている

結果：タスクの複雑さの影響

失敗したタスクの中でも、プロンプトの単語数が150を越えるタスクにおいて、ある特徴がみられた。

発見4：

短くて単純な問題と長くて複雑な問題との間に有意なパフォーマンスのギャップを観察した。

また、タスクプロンプトが150語を超える場合、出力の約60%はガベージコードである。これは、コード生成LLMが処理できるタスクの複雑さの上限を推定する研究の機会を浮き彫りにしている。

```
visited[grid[0][0]] = True
ans.append(grid[0][0])
ans.append(grid[0][1])
...
```

回答に関係ない操作を行っている

考察：LLMが生成したコードの障害特定

RQ1で、LLMは様々なコード構造(if, for,...)でエラーを発生させる可能性があることが示された

- しかし、そもそもエラーがなぜ生成されるのかはわからない
- 各デコードステップでのlogits, トークン確率分布, 自己注意スコア等, LLMは入出力以外にも様々な情報を持つ

LLMの各ステップにおけるパラメータを、エラーメッセージや他のコード特性とともに、モデルがどのステップで誤ったコードを生成し始めるかを予測するために活用できるかどうかを調査してみても面白いだろう。

考察：LLMが生成したコードのエラーの修復

RQ1で、LLMが犯すエラーは様々な種類に分けられることが示された。

- Ifエラーや不正確な引数のような単一行のエラーは、既存の自動プログラム修正(APR)技術が得意としている
 - 容易に修正できる
- しかし、LLMが犯すエラーは複数行のエラーであることが多いことが発見2で示された
 - 従来のAPR技術では、対応が困難

従来手法の限界を克服するために、LLMエージェントを用いたAPR技術が提案されている



LLMに与えるプロンプトに本研究の分類を元にしたエラー内容を与えれば、より正確に修正できるようになるのではないか？

考察：LLMが生成したコードの正しさの推定

発見 4 では、現在のLLMにおいても、複雑なタスク要件の理解および解決がまだ困難であることが示された

この観察から、2つの興味深い研究課題を提起できる

- 各LLMが処理できるタスクの複雑さの上限を推定することはできるのか？
- タスクの難易度と、そのタスクに対して生成されるコードの正しさを予想できるか？

結論

- 本研究は、LLMによって作成されたコード生成エラーに関する実証研究を提示している
- HumanEvalデータセットでの6つの人気のあるLLMの失敗ケースに基づいて、LLMのコード生成エラーの分類法を提案した
- 6つのLLMが、意味的および構文的エラーにおいて異なる分布を示すことを発見した
- バグ修正の労力、タスクの複雑さの影響とさまざまな種類のエラーとの相関関係を分析した

所感

「タスクの複雑さ」という曖昧すぎる概念

- 問題文が長かったら必ずしも複雑な問題なのか？
 - 京大の数学の入試問題は短くてもその解答は複雑
- 字面じゃなくて、意味的なスコアをとるべきなのかもしれない
 - できるのかはわからない

スライドのP29で提示した内容に疑問あり

- LLMのエラーにおいてGarbage codeが60%程度を占めていたとあった
- もしかするとGarbage Codeを出したLLMの大半はサイズの小さいモデル(Code-Gen-16B等)ではないのか？
- これが事実なら、あたかもGPT-4でも発生しているように聞こえる今回の表現は不適切なのかもしれない

補足：類似研究

2023年ごろに似たような研究が行われている[4]

- こちらは「バグ」の調査
- しかし、ここでいう「バグ」は本研究でいう「エラー」とほぼ同じ
 - 細かい分類が異なっているが、本質的に指しているものは同じと思われる
- 本研究と異なる点は以下の通りである
 - 評価対象のLLMが違う
 - 2023年の研究ではCodeGen, PanGu-Coder, Codexの3つ
 - 使用しているデータセットが違う
 - 2023年の研究は人間のバグとLLMのバグとの比較に焦点を置いている

[4]Tambon, F., Dakhel, A., Nikanjam, A., Khomh, F., Desmarais, M., & Antoniol, G. (2024). Bugs in Large Language Models Generated Code: An Empirical Study. *Empir. Softw. Eng.*, 30, 65.

(補足)本研究における意味的エラー

エラー特性	
条件エラー	条件が記述されていない
	条件が不適切
定数エラー	定数エラー
記述エラー	不適切なメソッド/変数
	定義されていない識別子
演算子/計算エラー	正しくない算術演算子
	正しくない比較演算子
不要なコード	対応するコードがないコメント
	役割がないコードスニペット
	意図と異なるロジック(wrong direction)
不完全なコード/ステップ	ステップが1つ足りていない
	ステップが複数足りていない
メモリエラー	無限ループ

(補足)本研究における構文エラー

エラー特性	
条件エラー	Ifエラー
Loopエラー	Forエラー
	Whileエラー
Returnエラー	正しくない返り値
メソッド(関数)呼び出しエラー	正しくない関数名
	正しくない引数
	呼び出すべきでない関数を持っている
代入エラー	正しくない定数
	正しくない算術計算
	正しくない変数名
	正しくない比較
Importエラー	Importエラー
コードブロックエラー	正しくないコードブロック
	あるべきコードブロックがない